JAIST Research Report IS-RR-98-0009S

Concurrent Object Composition in CafeOBJ

Shusaku Iida, Michihiro Matsumoto,[1]
Răzvan Diaconescu,[2] Kokichi Futatsugi,
and Dorel Lucanu[3]

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST).
[3]Faculty of Computer Science,
University "Al.I. Cuza" Iaşi, Romania.

[1]On leave from the Research Center of PFU Limited.
[2]On leave from the Institute of Mathematics of the Romanian Academy.

Concurrent Object Composition in CafeOBJ

Shusaku Iida, Michihiro Matsumoto,[1] Răzvan Diaconescu,[2]
Kokichi Futatsugi, and Dorel Lucanu[3]

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST).
[3]Faculty of Computer Science,
University "Al.I. Cuza" Iaşi, Romania.

Abstract

A new method is introduced to concurrently compose an object from already verified objects. The most important new feature of our method is that the verification of the composed object can be done by re-using the verifications of component objects. That is, the verification of composed object is also composable. This is not always true. We can show this can be achieved under some practically reasonable restrictions.

These can be made possible by using a new algebraic specification language CafeOBJ which has clear and precise algebraic semantics.

1 Introduction

The principle of "divide and conquer" seems to be the only effective principle in the development of large and complex systems. A system is divided into several independent components and each component is developed independently, after that the system is composed from the already developed components. In general this "divide and conquer" principle is applied recursively. Object-oriented modelling is widely used to support this compositional approach for system development. We also need methods which allow us to analyze and formally verify systems in the development of complex and critical systems; formal methods seem to suit this requirement. It is not so difficult to conclude that formal methods enhanced by object-oriented techniques can be used for the development of big, complex and critical systems which are the trend of current systems. There are several trials for this issue, for example, Object-Z[4] , FOOPS[1], etc. In this paper, we are going to introduce a new method

[1]On leave from the Research Center of PFU Limited.
[2]On leave from the Institute of Mathematics of the Romanian Academy.

for enhancing formal method with object-oriented techniques and show how to specify and verify systems in our method by using examples. We mainly focus on the modularity and the reusability power in object-oriented techniques. We use CafeOBJ [7, 19, 3, 5] as formal specification language.

Objects in CafeOBJ are treated with the hidden algebra formalism [11, 10], and the modularity is supported by the CafeOBJ module system. CafeOBJ handles module imports, parameterized modules, and general module expressions. By using hidden algebra, we can specify encapsulated objects and can handle highly abstracted specifications called behavioural specifications. We can prove system properties which are independent from the implementation of the system; we call these **behavioural properties**. We briefly explain hidden algebra in section 3.

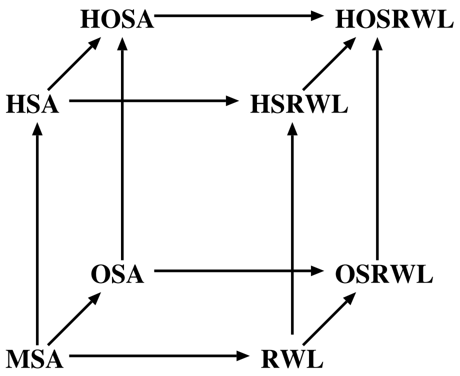
Reusability is the main issue of this paper. Generally, in object-oriented techniques, reusability is confined mainly to the reusability of source code. In this paper, we introduce a new notion of reusability that is **reusability of proofs**. We adopt object composition which supports both reusability of specification code and proofs. By using our method, we can start from valid small specifications which are relatively easy to handle and incrementally combine them to build the complete specification of the whole system. Proofs we can reuse in our method are behavioural equivalence proofs which are used for proving behavioural properties. For example, assume that we compose two objects to get an object for a system, and want to prove a behavioural property of the system. We need to define the behavioural equivalence for the composed object (the system) and need to prove that it is a really behavioural equivalence. How can we define the behavioural equivalence for the composed object? Do we need to prove a behavioural equivalence for the composed object each time we get a new object by composition? In this paper, we answer these question by showing that we can reuse the behavioural equivalence already proved for the composing objects.

In this paper, we firstly present CafeOBJ and hidden algebra, and then we present our method in section 4. After that we present some examples showing how to deal with dynamic and client-server systems in our method.

2 CafeOBJ

CafeOBJ [7, 19, 3, 5] is a multi-paradigm algebraic specification language which is a successor of OBJ[6, 13]. CafeOBJ is based on the combination of several logics consisting of **many sorted algebra**, **order sorted algebra** [9, 12], **hidden algebra**[11] and **rewriting logic**[16]. This combination is handled by **institutions**[8], as shown by the CafeOBJ **cube**[2] (see the figure). The arrows of the CafeOBJ cube correspond to institution embeddings. (**M** for **many**, **S** for **sorted**, **A** for **algebra**, **O** for **order**, **H** for **hidden** and **RWL** for **rewriting logic**.)

According to its semantics [2, 5], CafeOBJ can fit in several specification(and programming) paradigms such as equational specifications(and programmings), rewriting logic specification, behavioural concurrent specifications. Object orientation is a derived feature of CafeOBJ which can be treated both in behavioural specification and rewriting logic[15]; in this paper we consider only the behavioural specification approach. CafeOBJ has a power-



ful module system: several kinds of module imports, parameterized modules, and for each module one can choose between loose and tight (initial) semantics.

CafeOBJ is executable which means it can be used for rapid prototyping and theorem proving. Its operational semantics is based on term rewriting, and the proof calculi are equational and rewriting logic proof calculi. For confluent and terminating specification, two terms are equal when their normal forms are identical.

2.1 Syntax of CafeOBJ

Here, we briefly describe some basic syntax of CafeOBJ which is needed in this paper (we use hidden order sorted algebra but not rewriting logic). In this section we deal with some syntax related to order sorted algebra. The syntax related to hidden algebra can be found in section 3. The complete syntax of CafeOBJ can be found in [17]. Consider the following example, which specifies natural numbers:

```
mod! NAT {
-- declarations of sorts
  [ NzNat < Nat ]

-- declarations of operators
  op 0 : -> Nat
  op s : Nat -> NzNat
  op _+_ : Nat Nat -> Nat

-- declarations of variables
  vars N N' : Nat

-- declarations of equations
  eq 0 + N = N .
  eq s(N) + N' = s(N + N') .
}
```

The name of this module is `NAT` specified after the keyword `mod!` which is an abbreviation of `module!`. In CafeOBJ, the modules with tight (initial) semantics are declared by `module!`, and the modules with loose semantics are declared by `module*`. Sorts are declared within `[ ]` and the ordering of sorts are specified by using <. In this example, we have two sorts `NzNat` and `Nat`, and `NzNat` is a subsort of `Nat`. The lines beginning with the keyword `--` are comments. Operators are declared by the keyword `op` (`ops` for several operators with the same rank). The **arity** (a list of arguments) of an operator is specified before `->` and the sort (**coarity**) of an operator is specified after `->` (the pair of arity and coarity is called **rank**). Variables are declared by the keyword `var` (`vars` for several variables). Equations are declared by the keyword `eq` and conditional equations by `ceq`.

Modules can be imported by using `protecting`, `extending`, or `using`. `protecting` imports do not collapse elements or add new elements to the models of the imported module, but `extending` imports may add new elements but not collapse elements. In the folklore of algebraic specification these conditions are known under the name of "no junk and no confusion" and, respectively, "no confusion" condition. `Using` imports provide no guaranty, so they might even collapse elements. Every module implicitly imports the system module `BOOL` handling the Boolean data type. For confluent and terminating specifications, we can prove that two term are equal (that means two terms have the same normal form) by using the predicate == and system command `red` (abbreviation of `reduce`). For example, we can check the terms `s(s(0))` and `s(0) + s(0)` are the equal in the specification `NAT` in the following way:

```
NAT> red s(s(0)) == s(0) + s(0) .
-- reduce in NAT : s(s(0)) == s(0) + s(0)
true : Bool
(0.000 sec for parse, 3 rewrites(0.017 sec), 4 match attempts)
```

2.1.1 Some mathematical semantics:

A many sorted signature (S,Σ) consists of a set of sorts S and a set of S sorted operators Σ. An operator σ is denoted as $\sigma : w \to s$ where $w \in S^*$ is its **arity** and $s \in S$ is its **sort** (coarity). The **rank** of an operator consists of its arity and its sort. The set of operators of rank ws is denoted as Σ_{ws}. **Constants** are operations whose arity are empty, i.e., $\sigma :\to s$. In order sorted algebra, a signature is defined as $(S,\le,\Sigma)$, where $(S,\le)$ is a partial order set. A signature gives vocabularies for the sentences of a given specification. An equational specification SP is a pair consisting of signature and equations E for the signature. Signatures are sometimes denoted just as Σ, so equational specifications can be denoted as (Σ,E). A model (implementation) of a signature Σ is called **Σ-algebra** (we omit Σ when there is no confusion). Given a signature $(S,\le,\Sigma)$, an **algebra** A interprets

- each sort $s \in S$ as a set A_s(called the **carrier** of A of sort s),
- each subsort relation $s < s'$ as an inclusion $A_s \subseteq A_{s'}$, and
- each operator $\sigma \in \Sigma_{s_1 \ldots s_n s}$ as a function $A_\sigma : A_{s1} \times \ldots \times A_{s_n} \to A_s$.

Specifications and models are related by a satisfaction relation $\models$. An equation "$(\forall X)\, t = t'\ if\ C$" is satisfied by a Σ-algebra M, denoted as:

$$M \models (\forall X)\, t = t'\ if\ C$$

iff $\theta(t) = \theta(t')$ whenever $\theta(C) = M_{true}$ (the interpretation of the constant *true* in M) for all valuations $\theta : X \to M$, where X is a set of variables, t and t' are terms, C is the condition, and θ is an interpretation (homomorphism) which interprets terms to values of the model.

Given CafeOBJ signatures $(S, \leq, \Sigma)$ and $(S', \leq, \Sigma')$, then a **signature morphism** $\phi : (S, \leq, \Sigma) \to (S', \leq, \Sigma')$ consist of

- a mapping of sorts $f : S \to S'$ such that $f(s) \leq f(s')$ if $s < s'$, and
- an indexed family of mappings on operations, i.e., $(g_{s_1 \ldots s_n s} : \Sigma_{s_1 \ldots s_n s} \to \Sigma'_{f(s_1) \ldots f(s_n) f(s)})_{s_1, \ldots, s_n, s \in S, n \geq 0}$

For more details on the mathematical semantics of CafeOBJ see [3].

3 Hidden algebra

Specifications based on hidden algebra are called **behavioural specifications**[11, 10] and they can naturally handle states of encapsulated objects. The space of the states of an object is represented as a hidden sort which should be regarded as a kind of black box in the sense that we can observe the state of an object by using some operators called **attributes**.

In hidden algebra, there are two kind of sorts: visible and hidden. Visible sorts represent the data part of the specification and hidden sorts represent the states of objects. Given a signature $(S, \leq, \Sigma)$ with a subset $H \subset S$ of hidden sorts, a hidden model M (which can be either an algebra or rewriting model) interprets the visible sorts V and the operations Ψ of the visible sorts as a fixed model D (called the model of data), that is $M\restriction_{V,\Psi} = D$ (where $\restriction$ is the model reduct). Signature morphism $g : (S, H, \leq, \Sigma) \to (S', H', \leq, \Sigma')$ preserve the visible and hidden parts of the signatures, and obey the following conditions:

- g maps each behavioural operation to a behavioural operation,
- if $f(h) < f(h')$ for any hidden sorts h, h', then $h < h'$, and
- if $\sigma' \in \Sigma'_{w's'}$ is a behavioural operation and some sort in w' is hidden, then $\sigma' = g(\sigma)$ for some behavioural operation σ in Σ.

The last two conditions corresponding to object encapsulation conditions (see [10] for more details).

The following is an example of the behavioural specification of a counter of integers.

```
mod* COUNTER {
  protecting(INT)

  *[ Counter ]*

  op init : -> Counter                 -- initial state
  bop add : Int Counter -> Counter     -- method
  bop read : Counter -> Int            -- attribute

  var I : Int
  var C : Counter
```

```
  eq read(init) = 0 .
  eq read(add(I, C)) = I + read(C) .
}
```

The data of this behavioural specification is `INT` which is a built-in module of the system. It is imported in the specification by `protecting(INT)`. Hidden sorts are declared with `*[ ]*`. The keyword `bop` is used for behavioural operators. Behavioural operators have exactly a hidden sort in their arity, and when their sort is hidden they are called **methods** and when it is visible they are called **attributes**. In the above example, `add` is method and `read` is attribute.

Each sequence of methods determines an object state. In the above example, we can observe the state of `Counter` with reduce command (`red`) by using the attribute `read`:

```
COUNTER> red read(add(4, add(6, init))) .
-- reduce in COUNTER : read(add(4,add(6,init)))
10 : NzNat
(0.017 sec for parse, 5 rewrites(0.250 sec), 8 match attempts)
```

Behavioural specifications are based on loose semantics that means there exists several models (implementations) for them. For example, in `COUNTER` we can consider a model that keeps every history of methods applied (let's call this the **history model**) or a model that keeps just one integer value that is the result of the last applied method (let's call this the **cell model**). The following is the specification of the history model based on initial semantics (meaning that we only consider the initial model for the specification):

```
mod! COUNTER-HISTORY {
  protecting(INT)

  [ Counter ]

  op init : -> Counter
  op __ : Counter Counter -> Counter { assoc id: init}
  op __ : Int Counter -> Counter
  op add : Int Counter -> Counter
  op read : Counter -> Int

  vars I I' : Int
  var C : Counter

  eq add(I, C) = I C .
  eq read(init) = 0 .
  eq read(I C) = I + read(C) .
}
```

The following is the specification of the cell model based on initial semantics:

```
mod! COUNTER-CELL {
  protecting(INT)

  [ Counter ]

  op init : -> Counter
  op [_] : Int -> Counter
  op add : Int Counter -> Counter
  op read : Counter -> Int
```

```
  vars I I' : Int
  var C : Counter

  eq add(I, init) = [ I ] .
  eq add(I, [ I' ]) = [ I + I' ] .
  eq read(init) = 0 .
  eq read([ I ]) = I .
}
```

We can easily prove that models of above specifications are models of COUNTER too.

3.1 Behavioural properties

We want (all possible implementations of) counters to satisfy the following commutativity property of add (where `n:Int represent a term `n of sort Int):

```
add(`n:Int, add(`m:Int, init)) = add(`m, add(`n, init))
```

But this property does not hold in the history model within ordinary equational satisfaction (but in the cell model it does). These can be checked by using COUNTER-HISTORY and COUNTER-CELL.

```
COUNTER-HISTORY> red add(`n:Int, add(`m:Int, init)) == add(`m, add(`n, init)) .
-- reduce in COUNTER-HISTORY : add(`n:Int,add(`m:Int,init)) == add(
    `m:Int,add(`n:Int,init))
false : Bool
(0.000 sec for parse, 5 rewrites(0.017 sec), 5 match attempts)

COUNTER-CELL> red add(`n:Int, add(`m:Int, init)) == add(`m, add(`n, init)) .
-- reduce in COUNTER-CELL : add(`n:Int,add(`m:Int,init)) == add(`m:Int,
    add(`n:Int,init))
true : Bool
(0.033 sec for parse, 5 rewrites(0.000 sec), 19 match attempts)
```

We therefore need the commutativity of add as **behavioural equivalence** rather than strict equivalence. The intuitive understanding of behavioural equivalence is that two states are behavioural equivalent when they cannot be distinguished under all the observations (by using all the attributes) after applying any method.

The **behavioural equivalence** denoted as $\equiv$ can be defined as follows [11]:

- when $s \in V$:
  $a \equiv a'$ iff $a = a'$
- when $s \in H$:
  $a \equiv a'$ iff $c(a) = c(a')$ for all $v \in V$ and for all visible contexts c.

where $a, a' \in A_s$, V is the set of visible sorts, H is the set of hidden sorts, a **context** is a term which is a sequence of behavioural operators, and a **visible context** is a context of visible sort. In CafeOBJ behavioural equivalence is denoted by the special keyword beq (bceq for the conditional case).

Proving behavioural equivalence by directly using its definition means a proof by induction on the structure of contexts; this is called **context induction**[14]. For large specification, context induction can lead to very complex proofs. The **coinduction** method [11] avoids such problems. Correctness of coinduction is based on the following theorem[11]:

Theorem 1 *Behavioural equivalence is the largest hidden congruence (congruence with respect to behavioural operations).* □

A proof by coinduction consists of the following steps:

1. give a **candidate** hidden congruence relation R
2. prove that R is a hidden congruence for all the behavioural operators,
3. prove the behavioural property by using R.

The following is a coinduction proof for the behavioural commutativity of add for the specification COUNTER:

```
open COUNTER
op _R_ : Counter Counter -> Bool .

-- give a candidate of hidden congruence relation
vars C1 C2 : Counter
eq C1 R C2 = read(C1) == read(C2) .

-- hypothesis
ops c1 c2 : -> Counter .
eq read(C1) = read(C2) .

-- prove the R is a congruence
op i : -> Int .
red add(i, c1) R add(i, c2) .

-- prove beq add(n:Int, add(m:Int, init)) = add(m, add(n, init)) .
red add(n:Int, add(m:Int, init)) R add(m, add(n, init)) .
close
```

In many cases the following relation:

$$t =\!*\!= t' \; iff \; \bigwedge_a a(t) == a(t') \text{ for all the attributes } a$$

where t, t' are terms of (the same) hidden sort, is a hidden congruence, therefore it can be used as the candidate hidden congruence relation. CafeOBJ adopts this as a default coinduction relation and the system provides automatic support for proving it is a congruence. In the case of COUNTER, this mechanism succeeds so the proof of commutativity of add consists of just the following reduction:

```
CafeOBJ> in counter
-- processing input : ./counter.mod
-- defining module* COUNTER......._..*
** system already proved =*= is a congruence of COUNTER done.

COUNTER> red add('n:Int, add('m:Int, init)) =*= add('m, add('n, init)) .
-- reduce in COUNTER : add('n:Int,add('m:Int,init)) =*= add('m:Int,
    add('n:Int,init))
true : Bool
(0.033 sec for parse, 8 rewrites(0.067 sec), 48 match attempts)
```

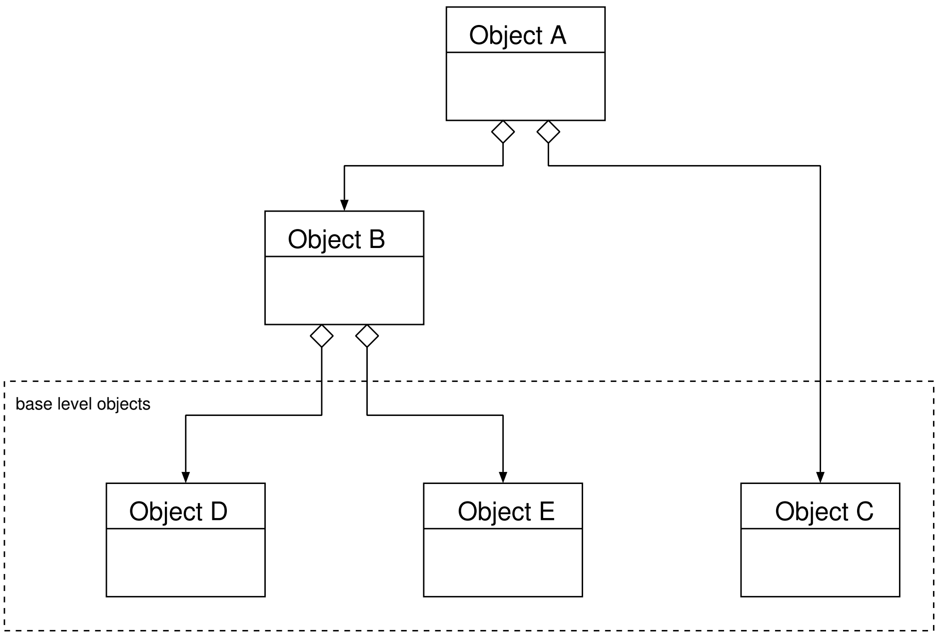
4 Reusability

One of the most important issue in object-oriented techniques is reusability. In object-oriented programming, reusability of the source code is important. But, in object-oriented specification, reusability of the proofs is also very important because of the verification process. In this section, we are going to present a new method to reuse both specification code and proofs.

There seems to be two different techniques to reuse code: composition and inheritance. Our method belongs more to the object composition side. This paper promotes the view that composition is more effective than inheritance as reusability techniques. We will first present our composition method and afterwards we will compare it with the inheritance method in CafeOBJ .

4.1 Object composition

The following figure represents the structure of object compositions by using OMT[18] like notation to represent relations between objects.



Reusing specifications is done by the **projection operators**. Projection operators are defined for each composing objects to get their states from the state of composed object. All methods of the composed object are related to the methods of the composing objects using these projection operators.

Definition 1 *An operator* $\pi_n : h \to h_n$ *is a projection operator if:*

1. *h is a hidden sort of the composed object O,*

2. h_n *are hidden sorts of the composing objects* O_n*,*

3. *for each attribute a of O, there exists a composing object O_n, an operator $f : v_{n_1} ... v_{n_m} \to v$ (v_{n_m} is a data for O_n and v is a visible sort), and a visible O_n-context c_n such that:* [1]
   $a = (\pi_{n_1}; c_{n_1}, ..., \pi_{n_m}; c_{n_m}) f$,

4. *for each method $m : h \to h$ of O, for all composing object O_n, there exists a sequence of methods m_n such that:*
   $m; \pi_n = \pi_n; m_n$,

5. *for each constant const $:\to h$ of O, for all composing objects O_n, there exists a constant $const_n :\to h_n$ such that:*
   $const; \pi_n = const_n$

□

This definition is for static systems (i.e., configuration of the system is unchanged when it is running), see section 5 for dynamic systems.

Notice that the equalities defining the attributes and methods for the composed objects are strict (i.e., not behavioural) equations and that projection operators can be either ordinary or behavioural. Using non-behavioural projections has the advantage to enable a user controlled selection of the attributes on the composed object, but they have the disadvantage of possibly restricting the computations involving behavioural equations. [2]

Definition 2 *As shown in the above figure, the structure of such a composition is a DAG (directed acyclic graph). A* **base level object** *is an object without projection operators.* □

Definition 3 *Two methods of a composed object are in the same* **method group** *when they are related to the same composing object.* □

If a method in a composed object relate to several method in different composing objects then there is a overlapping among the method groups.

Object composition can be classified with respect to how the composing objects are connected: concurrent connection and synchronized concurrent connection. We are going to discuss both cases in the following subsections.

4.1.1 Concurrent connection:

In the case of concurrent connection, we have full concurrency between all the composing objects. This means that if two methods are in the different method groups then all states containing these methods (i.e. possibly in different order) are behavioural equivalent.

For example, assume that we want to compose two counters, `Counter1` and `Counter2` and get `2Counter`. Both the composing objects are equal to `COUNTER` we showed previously, just the sort name is renamed to `Counter1` and `Counter2`, respectively. `2Counter` has two methods `add1` and `add2` to count up `Counter1` and `Counter2`, respectively. So, there is no intersection between `Counter1` and `Counter2`. The following is the specification of `2Counter`:

[1] We use the diagrammatic notation for the composition of functions.

[2] See [3] for details on reductions involving behavioural equations. However, the projections are **behavioural coherent** [3] we have the same computational power as in the case of behavioural projections.

```
mod* 2COUNTER {
  protecting(COUNTER *{ hsort Counter -> Counter1,
                       op init -> init1 })
  protecting(COUNTER *{ hsort Counter -> Counter2,
                       op init -> init2 })

  *[ 2Counter ]*

  op init : -> 2Counter                -- initial state
  bop add1 : Int 2Counter -> 2Counter  -- method
  bop add2 : Int 2Counter -> 2Counter  -- method
  bop counter1 : 2Counter -> Counter1   -- projection
  bop counter2 : 2Counter -> Counter2   -- projection

  var I : Int
  var TC : 2Counter

  eq [c1-1] : counter1(init)        = init1 .
  eq [c1-2] : counter1(add1(I, TC)) = add(I, counter1(TC)) .
  eq [c1-3] : counter1(add2(I, TC)) = counter1(TC) .

  eq [c2-1] : counter2(init)        = init2 .
  eq [c2-2] : counter2(add1(I, TC)) = counter2(TC) .
  eq [c2-3] : counter2(add2(I, TC)) = add(I, counter2(TC)) .
}
```

Firstly, we have to import twice `COUNTER` by renaming its hidden sort so that each of them have a different sort name. Secondly, we define a new hidden sort and operators for the composed object. Finally, we define projection operators and equations for them. CafeOBJ syntax allows us to put labels to equations, like `[c1-1]` for the first equation in the above specification. Equation `[c1-3]` and `[c2-2]` express the concurrency of the composing counters.

There are two method groups with respect to the two composing objects: `Counter1` and `Counter2`. Methods `add1` and `add2` are in the different method group, so `add1` and `add2` can be operated concurrently. For example, we can prove the following behavioural property:

```
beq add1(i1, add2(i2, init)) = add2(i2, add1(i1, init)) .
```

We are going to present the details of this proof in section 4.2.

4.1.2 Synchronized concurrent connection:

In the case of synchronized concurrent connection, the concurrency between composing objects is partial (i.e. some synchronizations happens). Synchronization happens when:

1. the projected state of the composed object (via a projection operator) depends on the state of a different (from the object corresponding to the projection operator) composing object,
2. methods of the composed object change simultaneously states of several composing objects

These conditions amount to refining Definition 1 by considering conditions for the projection operator of the composed object.

Definition 4 *The conditions for these (conditional) equations should fulfill the following:*

- *each condition is a finite conjunction of equalities between terms of the form* $\pi_n; c_n$ *(where* π_n *is a projection operator and* c_n *is an* O_n*-context) and terms in the data signature, and*

- *disjunction of all the conditions corresponding to a given left hand side is always true.*

□

Here, we consider a special counter with switch, which has a method `put` to add or subtract a natural number to (or from) the counter. We, again, reuse the specification of the counter we used before. Note that the interface of counter and counter with switch is different (method `add` in `COUNTER` takes an integer number but here `put` takes a natural number). The composing objects are: `Switch` and `Counter`. The method `put` in the composed object counts up the counter if the switch is on and count down if the switch is off.

Firstly, we specify a switch as follows:

```
mod! ON-OFF {
  [ Value ]

  ops on off : -> Value
}

mod* SWITCH {
  protecting(ON-OFF)

  *[ Switch ]*

  op init : -> Switch
  bop on_ : Switch -> Switch      -- method
  bop off_ : Switch -> Switch     -- method
  bop state_ : Switch -> Value    -- attribute

  var S : Switch

  eq state(init) = off .
  eq state(on(S)) = on .
  eq state(off(S)) = off .
}
```

The following is the specification of the counter which switch:

```
mod* COUNTER-WITH-SWITCH {
  protecting(COUNTER + SWITCH)

  *[ Cws ]*

  op init-cws : -> Cws              -- initial state
  bop add : Cws -> Cws              -- method
  bop sub : Cws -> Cws              -- method
  bop put : Nat Cws -> Cws          -- method
```

```
  bop read : Cws -> Int              -- attribute
  bop counter_ : Cws -> Counter      -- projection
  bop switch_ : Cws -> Switch        -- projection

  var N : Nat
  var C : Cws

  eq read(C) = read(counter(C)) .

  eq [s-1] : switch(init-cws) = init .
  eq [s-2] : switch(put(N, C)) = switch(C) .
  eq [s-3] : switch(add(C)) = on(switch(C)) .
  eq [s-4] : switch(sub(C)) = off(switch(C)) .

  eq  [c-1] : counter(init-cws) = init .
  ceq [c-2] : counter(put(N, C)) = add(N, counter(C))
              if state(switch(C)) == on .
  ceq [c-3] : counter(put(N, C)) = add(-(N), counter(C))
              if state(switch(C)) == off .
  eq  [c-4] : counter(add(C)) = counter(C) .
  eq  [c-5] : counter(sub(C)) = counter(C) .
}
```

Synchronization can be seen in `[c-2]` and `[c-3]` which corresponds to the first synchronization case (i.e., the definition of the counter depends on the state of the switch).

4.2 Verification of a composed object

As we described in section 3, our concern is mainly with behavioural properties. Behavioural properties for base level objects can be proved by using coinduction. In many cases, base level objects are simple and small so it is easy to prove the behavioural equivalence for them. Behavioural equivalence in composed objects is a conjunction of all the behavioural equivalence of composing objects.

Theorem 2 *Given the states s and s' of a composed object, then:*

$$(s \equiv s') \textit{ if } \bigwedge_{n \in CObj} (\pi_n(s) \equiv_n \pi_n(s'))$$

where $\equiv$ is the behavioural equivalence in the composed object, $CObj$ is a set of composing objects, $\equiv_n$ is the behavioural equivalence of the composing object O_n, and π_n is the projection operator to the composing object O_n. $\Box$

The proof of this theorem is in appendix A.

Corollary 1 *If all projection operators are behavioural then*

$$(s \equiv s') = \bigwedge_{n \in CObj} (\pi_n(s) \equiv_n \pi_n(s'))$$

In the counter with switch example, the behavioural equivalence of composing objects is just the default coinduction relation and automatically provided by the CafeOBJ system. So, from the above theorem, we can reuse the proofs of behavioural equivalence of the composing objects and get the behavioural equivalence of counter with switch.

```
op _R_ : Cws Cws -> Bool .
vars C1 C2 : Cws
eq C1 R C2 = switch(C1) =*= switch(C2) and counter(C1) =*= counter(C2) .
```

For example, by using this behavioural equivalence, we can prove the following behavioural property:

```
-- reduce in % : put(m,add(put(n,sub(init-cws)))) R add(put(n,sub(
    put(m,add(init-cws)))))
true : Bool
(0.033 sec for parse, 68 rewrites(0.050 sec), 198 match attempts)
```

Notice that crucial role played by the `add` at the top of the right hand side of the previous property, since without it the `SWITCH` object would be in behaviourally non-equivalent states.

It is also easy to prove the behavioural property explained in section 4.1.1 by reusing the proof of behavioural equivalence `=*=` in `COUNTER`. The following is the proof score for this:

```
op _R_ : 2Counter 2Counter -> Bool .
vars C1 C2 : 2Counter
eq C1 R C2 = counter1(C1) =*= counter1(C2) and counter2(C1) =*= counter2(C2) .

ops i1 i2 : -> Int .
red add1(i1, add2(i2, init)) R add2(i2, add1(i1, init)) .
```

4.3 Correctness proof for composition

This is based on the idea that a composition is correct when the composed object is the refinement of its components and for the concurrent part the commutativity equations corresponding to the concurrency of methods/attributes belonging to different components hold. This follows some early work on concurrent composition of [10].

For example, we can show that the `COUNTER-WITH-SWITH` is a correct composition of `COUNTER` and `SWITCH` as follows. In order to express properly the morphisms used in the refinement proof, we need the following "derived" method:

```
bop addc : Int Cws -> Cws

ceq addc(I, C) = put(I, C) if state(switch C) == on .
ceq addc(I, C) = put(-(I), C) if state(switch C) == off .
```

For proving that counter with switch is a correct composition of `SWITCH` and `COUNTER`, we define the following "synchronization morphism": [3]

- ψ_1 : `SWITCH` $\to$ `COUNTER-WITH-SWITCH` such that:

$$
\begin{aligned}
\psi_1(\texttt{init}) &= \texttt{init-cws} \\
\psi_1(\texttt{on}) &= \texttt{add} \\
\psi_1(\texttt{off}) &= \texttt{sub} \\
\psi_1(\texttt{state}) &= \texttt{switch;state}
\end{aligned}
$$

[3] see [10] for the mathematical definition of synchronization morphism.

- ψ_2 : `COUNTER` $\to$ `COUNTER-WITH-SWITCH` such that:

$$\begin{aligned}\psi_2(\texttt{init}) &= \texttt{init-cws}\\ \psi_2(\texttt{add}) &= \texttt{addc}\\ \psi_2(\texttt{read}) &= \texttt{read}\end{aligned}$$

We prove that `COUNTER-WITH-SWITCH` refines `SWITCH` via ψ_1:

```
red state switch add(c) == on .
red state switch sub(c) == off .
```

We prove that `COUNTER-WITH-SWITCH` refines `COUNTER` via ψ_2:

```
--> case 1:
eq state(switch c) = on .
red read addc(i, c) == i + read c .
--> case 2:
eq state(switch c) = off .
red read addc(i, c) == i + read c .
```

We prove the commutativity equations corresponding to the methods.

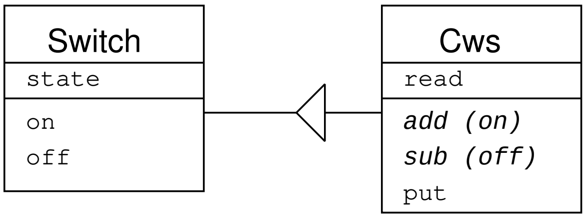
```
--> case 1:
eq state(switch c) = on .
red add(addc(i, c)) R addc(i, add(c)) .
red sub(addc(i, c)) R addc(i, sub(c)) .
--> case 2:
eq state(switch c) = off .
red add(addc(i, c)) R addc(i, add(c)) .
red sub(addc(i, c)) R addc(i, sub(c)) .
```

Finally, we have commutativity equations corresponding to the attributes.

```
red state(switch put(i, c)) == state(switch c) .
red read(counter add(c)) == read(counter c) .
red read(counter sub(c)) == read(counter c) .
```

4.4 Inheritance

In hidden algebra, inheritance is modelled via subsort relations [10] (we only consider a single inheritance case). This means that the space of states of the inheriting object is included in the space of states of the inherited object; this enables the inherited methods to act on the states of the inheriting object. Let's consider the counter with switch example we used before. We can build it differently by inheriting `Switch` (`add` and `sub` being just the renamings of `on` and `off` and using essentially the same names for operators as in the previous specification).



```
mod* COUNTER-WITH-SWITCH {
  protecting(INT)
  protecting(SWITCH *{ bop on_ -> add_,
                       bop off_ -> sub_ })

  *[ Cws < Switch ]*

  op init-cws : -> Cws        -- initial state
  bop add_ : Cws -> Cws       -- method
  bop sub_ : Cws -> Cws       -- method
  bop put  : Nat Cws -> Cws   -- method
  bop read : Cws -> Int       -- attribute

  var C : Cws
  var N : Nat

  eq state(init-cws) = state(init) .
  eq state(put(N, C)) = state(C) .

  eq read(init-cws) = 0 .
  eq read(add(C)) = read(C) .
  eq read(sub(C)) = read(C) .
  ceq read(put(N, C)) = N + read(C)
      if state(C) == on .
  ceq read(put(N, C)) = -(N) + read(C)
      if state(C) == off .
}
```

Notice that in the inheritance approach we can reuse the proofs of the coinduction relations for the inherited sorts since no new methods/attributes can be added on the inherited object. In this example, any coinduction relation R has two components $R_{\mathtt{Cws}}$ and $R_{\mathtt{Switch}}$ satisfying $R_{\mathtt{Cws}} \subseteq R_{\mathtt{Switch}}\restriction_{\mathtt{Cws}\times\mathtt{Cws}}$. The congruence proof for R is therefore necessary only for $R_{\mathtt{Cws}}$. However, in this example, behavioural equivalence is again the default coinduction relation =*=. [4]

If we compare the composition and inheritance approaches, we notice that (single) inheritance can be regarded as "sequential composition". This would be more obvious if one thinks of an example composing three objects, then in the inheritance approach one need two inheritance levels.

5 Specification of dynamic systems

Dynamic systems are different from static systems in that the configuration of the system changes when the system is running. The key point is that we need some kind of identifiers to manage object creation and deletion.

Definition 5 *An* **dynamic object** *can be created or deleted in a composed object and its initialization is done with appropriate data playing the role of object identifier.* □

The definition of projection operators for dynamic objects are the same as Definition 1 except that a projection operator for a dynamic object has an object identifier and a composed object as its arity as follows:

[4]The current CafeOBJ implementation does not have a mechanism to reuse the "inherited" components of =*=.

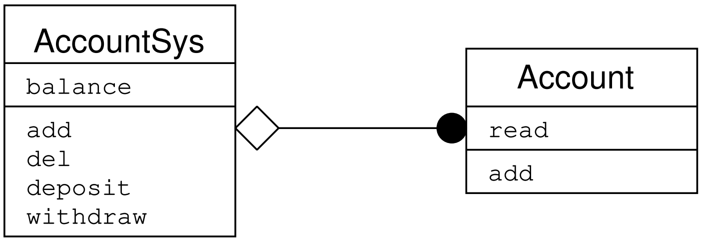
Definition 6 *An operator $\pi_n : ID_n h \to h_n$ is a projection operator if*

1. *h is a hidden sort of the composed object O,*
2. *h_n are hidden sorts of the composing objects (the index n corresponding to the same class of objects),*
3. *ID_n is the set of identifiers for the objects of class n, and*
4. *fulfill the last three conditions(3, 4, 5) of Definition 1.*

□

Static systems appear as dynamic systems in which classes have a fixed number of objects. For example, the counter with switch example has two classes (for the composing objects) with one object each. The following example, has only one class (for the composing objects) but with a dynamic numbers of objects.

For an example of dynamic systems specification, we use a specification of a bank account system. A bank account system consists of several individual accounts of customers. It has methods: `add` for adding a new account, `del` for deleting an existing account, `deposit`, `withdraw`, and an attribute `balance`.



We again reuse the specification of counter for `Account` with the difference that the initial state of the counter is indexed by the user identifiers. User identifiers which are used for the object identifier of the new `Counter` are specified as follows:

```
mod! USER-ID {
  extending(NAT *{ sort Nat -> UId })

  op unidentified-user : -> UId
}
```

The module `COUNTER` and `COUNTER*` (which contains an error value) can be specified as follows:

```
mod* COUNTER {
  protecting(USER-ID + INT)

  *[ Counter ]*

  -- initialize counter with user ID
  op init-counter : UId -> Counter
  -- add a value to the counter (method)
  bop add : Int Counter -> Counter
```

```
  -- read the value of the counter (attribute)
  bop read_ : Counter -> Int

  var I : Int
  var C : Counter
  var U : UId

  eq read(init-counter(U)) = 0 .
  eq read(add(I, C)) = I + read(C) .
}

mod* COUNTER* {
  protecting(COUNTER)

  -- error value
  op counter-not-exist : -> Counter  -- error
}
```

The following is the specification of a bank account system (`Account` can be obtained by renaming `COUNTER*`):

```
mod* ACCOUNT-SYSTEM {
  protecting(COUNTER* *{ hsort Counter -> Account,
                         op init-counter -> init-account,
                         op no-counter -> no-account })

  *[ AccountSys ]*

  op init-account-sys : -> AccountSys              -- initial state
  bop add : UId Nat AccountSys -> AccountSys       -- method
  bop del : UId AccountSys -> AccountSys           -- method
  bop deposit : UId Nat AccountSys -> AccountSys   -- method
  bop withdraw : UId Nat AccountSys -> AccountSys  -- method
  bop balance : UId AccountSys -> Nat              -- attribute
  bop account : UId AccountSys -> Account          -- projection

  vars U U' : UId
  var A : AccountSys
  var N : Nat

  eq account(U, init-account-sys) = no-account .
  ceq account(U, add(U', N, A)) = add(N, init-account(U))
     if U == U' .
  ceq account(U, add(U', N, A)) = account(U, A)
     if U =/= U' .
  ceq account(U, del(U', A)) = no-account
     if U == U' .
  ceq account(U, del(U', A)) = account(U, A)
     if U =/= U' .
  ceq account(U, deposit(U', N, A)) = add(N, account(U, A))
     if U == U' .
  ceq account(U, deposit(U', N, A)) = account(U, A)
     if U =/= U' .
  ceq account(U, withdraw(U', N, A)) = add(-(N), account(U, A))
     if U == U' .
  ceq account(U, withdraw(U', N, A)) = account(U, A)
     if U =/= U' .

  eq balance(U, A) = read(account(U, A)) .
}
```

We can get the state of an individual account by using the projection operator `account` by specifying a user identifier. User identifiers work as object identifiers. The attribute `balance` is just a abbreviation of `read` of an individual account. Let's consider the following example. We make a new account with user identifier `'u` and initial balance `10` and deposit `4` then withdraw `2` and observe the balance of the account.

```
red balance('u:UId, withdraw(2, deposit(4, add('u, 10, init-account-
sys)))) .
```

The following is the result of the above reduction:

```
-- reduce in ACCOUNT-SYSTEM : balance('u:UId,withdraw('u:UId,2,deposit(
    'u:UId,4,add('u:UId,10,init-account-sys))))
12 : NzNat
(0.010 sec for parse, 19 rewrites(0.020 sec), 54 match attempts)
```

5.1 Verification of dynamic systems

The behavioural equivalence of a composed object is a conjunction of all the behavioural equivalence of the composing objects. This refines Theorem 2 for the case of dynamic systems, thus giving the possibility to reuse the behavioural equivalence of the composing objects.

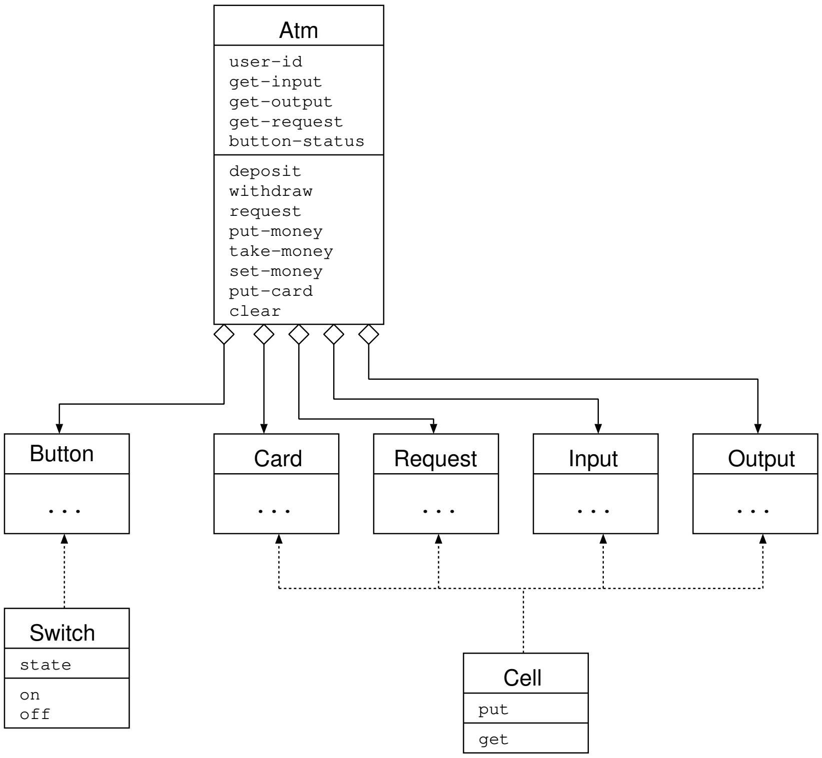
Corollary 2 *Given the states s and s' of a composed object, then:*

$$(s \equiv s')\ \mathit{if} \bigwedge_{n \in CClass} \Big(\bigwedge_{i \in ID_n} (\pi_n(i,s) \equiv_n \pi_n(i,s')) \Big)$$

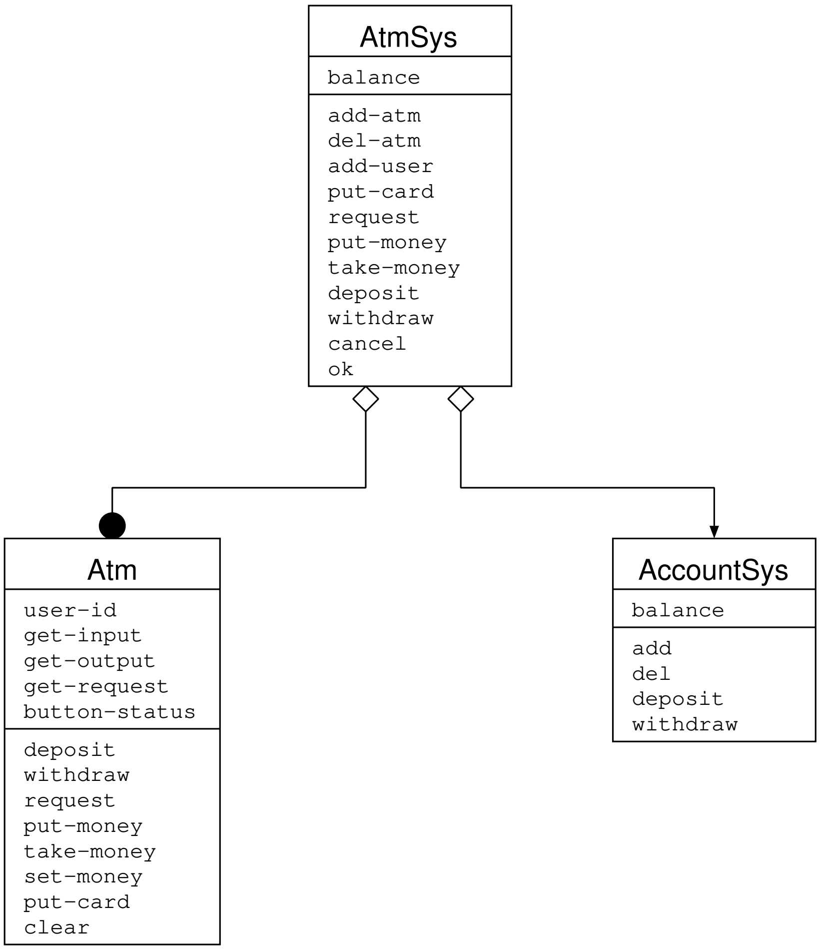
where $\equiv$ is the behavioural equivalence of the composed object, CClass is the set of classes of objects, $\equiv_n$ is the behavioural equivalence for the class n, and ID_n and π_n have the same meaning as in Definition 1. If all projection operators are behavioural then the other implication holds too. □

6 Specification of client-server systems

In this section, we consider an ATM system which consists of several ATM clients and a bank account (a complete specification of the ATM system can be found in Appendix B). An ATM client can be specified by composing a switch (buttons for selecting deposit or withdraw) and four cells to hold card information, request for withdraw, input money, and output money (all the objects are composed by concurrent connection). We omit the CafeOBJ ATM specification and just show it in the following figure (the renamings of objects are represented by dotted arrows):



Now we have all objects needed to specify an ATM system. ATM clients and the bank account are composed into an ATM system by synchronized concurrent connection. The following figure shows the ATM system:



The following is the signature part of an ATM system in CafeOBJ . Two projection operators are defined: `account-sys` for getting the state of a bank account and `atm` for individual atm clients.

```
mod* ATM-SYSTEM {
  protecting(ACCOUNT-SYSTEM + ATM-CLIENT)

  *[ System ]*

  op init-sys : -> System                        -- initial state
  bop add-atm : AId System -> System             -- method
  bop del-atm : AId System -> System             -- method
  bop add-user : UId Nat System -> System        -- method
  bop put-card : AId UId System -> System        -- method
  bop request : AId Nat System -> System         -- method
  bop put-money : AId Nat System -> System       -- method
  bop take-money : AId System -> System          -- method
  bop deposit : AId System -> System             -- method
  bop withdraw : AId System -> System            -- method
  bop ok : AId System -> System                  -- method
  bop cancel : AId System -> System              -- method
  bop balance : UId System -> Nat                -- attribute
  bop account-sys : System -> AccountSys         -- projection
  bop atm : AId System -> Atm                    -- projection
```

In the case of client-server system, we need a communication mechanism between the clients and the server. This can be achieved by using a server method having attributes of client objects as arguments. Assume that a customer pushed an ok button (i.e. the method `ok` is applied). Then informations in the ATM clients are sent to the server and the state of a bank account changes accordingly to these informations and the state of a bank account. This can be specified in CafeOBJ as follows:

1. card is inputed and `deposit` button is pushed:

```
ceq account-sys(ok(A, S)) =
     deposit(user-id(atm(A, S)),
             get-input(atm(A, S)),
             account-sys(S))
     if button-status(atm(A, S)) == deposit and
        user-id(atm(A, S)) =/= unidentified-user and
        get-input(atm(A, S)) =/= 0 .
```

2. card is inputed, `withdraw` button is pushed, and the request for withdraw is less or equal than the balance of the user:

```
ceq account-sys(ok(A, S)) =
     withdraw(user-id(atm(A, S)),
              get-request(atm(A, S)),
              account-sys(S))
     if button-status(atm(A, S)) == withdraw and
        user-id(atm(A, S)) =/= unidentified-user and
        get-request(atm(A, S)) =/= 0 and
        get-request(atm(A, S)) <=
               balance(user-id(atm(A, S)),
                       account-sys(S)) .
```

3. card is inputed, `withdraw` button is pushed, and the request for withdraw is greater than the balance of the user:

```
ceq account-sys(ok(A, S)) = account-sys(S)
     if user-id(atm(A, S)) == unidentified-user or
         (button-status(atm(A, S)) == deposit and
             get-input(atm(A, S)) == 0) or
         (button-status(atm(A, S)) == withdraw and
             (get-request(atm(A, S)) == 0 or
             get-request(atm(A, S)) >
                      balance(user-id(atm(A, S)),
                              account-sys(S)))) .
```

For example, in the second case of above equation, the user identifier held (recorded) in the ATM client (`user-id(atm(A, S))`) and the amount of requested money (`get-request(atm(A, S))`) are both sent to the bank account by the method `withdraw` if all conditions are satisfied. If conditions are not satisfied then the data is not sent to the server (see the third equation above).

7 Verification of ATM system

In this section, we are going to prove a bihavioural property of ATM system. First, we define a module called `ATM-SYSTEM-TOPLEVEL` which provides basic interface of ATM system like `withdraw` or `deposit`.

```
mod* ATM-SYSTEM-TOPLEVEL {
  protecting(ATM-SYSTEM)

  *[ TopLevel ]*

  op init-tl : -> TopLevel                            -- initial state
  bop add-atm : AId TopLevel -> TopLevel              -- method
  bop del-atm : AId TopLevel -> TopLevel              -- method
  bop add-user : UId Nat TopLevel -> TopLevel         -- method
  bop del-user : UId TopLevel -> TopLevel             -- method
  bop deposit : UId AId Nat TopLevel -> TopLevel      -- method
  bop withdraw : UId AId Nat TopLevel -> TopLevel     -- method
  bop balance : UId TopLevel -> Nat                   -- attribute
  bop system : TopLevel -> System                     -- projection

  var U : UId
  var A : AId
  var N : Nat
  var TL : TopLevel

  eq balance(U, TL) =
     balance(U, account-sys(system(TL))) .

  eq system(init-tl) = init-sys .
  eq system(add-atm(A, TL)) = add-atm(A, system(TL)) .
  eq system(del-atm(A, TL)) = del-atm(A, system(TL)) .
  eq system(add-user(U,N,TL)) = add-user(U,N,system(TL)) .
  eq system(del-user(U, TL)) = del-user(U, system(TL)) .
  eq system(deposit(U, A, N, TL)) =
     ok(A, put-money(A, N,
       deposit(A, put-card(A, U, system(TL))))) .
  eq system(withdraw(U, A, N, TL)) =
     take-money(A, ok(A, request(A, N, withdraw(A,
                 put-card(A, U, system(TL)))))) .
}
```

The following module defines the behavioural equivalence for the whole system by using the results about the reusability of behavioural equivalence of the composing objects and the CafeOBJ default coinduction relation for the base level objects since for all these the default coinduction checking succeeds.

```
mod COINDUCTION-REL {
  protecting(ATM-SYSTEM-TOPLEVEL)
```

The following is the behavioural equivalence relation for `ACCOUNT-SYSTEM` which is parameterized by the user identifiers; and which reuses the behavioural equivalence on `COUNTER*` which is the default coinduction relation `=*=`.

```
  op _R[_]_ : AccountSys UId AccountSys -> Bool {coherent}

  vars AS1 AS2 : AccountSys
  var U : UId

  eq AS1 R[U] AS2 = account(U, AS1) =*= account(U, AS2) .
```

The behavioural equivalence on the `ATM-CLIENT` is the conjunction of the behavioural equivalences of its composing objects; all of these are the default coinduction relations `=*=`.

```
  op _R_ : Atm Atm -> Bool {coherent}

  vars A1 A2 : Atm

  eq A1 R A2 = button(A1) =*= button(A2) and
               card(A1) =*= card(A2) and
               request(A1) =*= request(A2) and
               input(A1) =*= input(A2) and
               output(A1) =*= output(A2) .
```

The behavioural equivalence for ATM-SYSTEM is the conjunction of the behavioural equivalences for ACCOUNT-SYSTEM and the conjunction of behavioural equivalences for all ATM clients
(ATM-CLIENT).

```
  op _R[_,_]_ : System UId AId System -> Bool {coherent}

  vars S1 S2 : System
  var A : AId

  eq S1 R[U, A] S2 = account-sys(S1) R[U] account-sys(S2)
                     and  atm(A, S1) R    atm(A, S2) .
```

Finally, the behavioural equivalence at the top level is just the behavioural equivalence of the system.

```
  op _R[_,_]_ : TopLevel UId AId TopLevel
                                   -> Bool {coherent}

  vars T1 T2 : TopLevel

  eq T1 R[U, A] T2 = system(T1) R[U, A] system(T2) .
}
```

Now, we can proceed to do the proof of a behavioural property stating the true concurrency of cash withdrawals by different users without respect of the ATM machines involved or the amount of cash requested. At the top level this property can be expressed as:

$$\texttt{withdraw}(u_1, A_1, N_1, \texttt{withdraw}(u_2, A_2, N_2, state)) \sim$$
$$\texttt{withdraw}(u_2, A_2, N_2, \texttt{withdraw}(u_1, A_1, N_1, state))$$

where u_i are users (identifiers), A_i are ATM machines (identifiers), and N_i are amounts of cash requested for withdrawal, for $i \in \{1,2\}$. Other parameters for this proof are the balance M_i of the accounts of the users u_i, for $i \in \{1,2\}$. The relationship between all these parameters amounts to a complex case analysis involving 108 cases. Fortunately, these can be automatically generated by CafeOBJ via a suitable meta-level encoding:

```
mod PROOF {
  protecting(COINDUCTION-REL)

  ops a a1 a2 : -> AId
  op t : -> TopLevel
  ops u u1 u2 : -> UId
  ops n1 n2 n01 n02 m1 m1' m2 m2' : -> Nat
```

Case analysis with respect to balances of accounts and requested amounts:

```
eq n1 =/= 0 = true .
eq n2 =/= 0 = true .
eq n01 == 0 = true .
eq n02 == 0 = true .
eq n1 <= m1 = true .
eq n01 <= m1 = true .
eq n1 > m1' = true .
eq n2 <= m2 = true .
eq n02 <= m2 = true .
eq n2 > m2' = true .
```

The following operations and equations generate the final proof term (`RESULT`) which includes all cases generated by the case analysis:

```
op state-of-system : Nat Nat -> TopLevel
ops w1w2 w2w1 : AId AId Nat Nat Nat Nat -> TopLevel
op TERM : UId AId AId AId Nat Nat Nat Nat -> Bool
op TERM1 : UId AId AId AId Nat Nat -> Bool
op TERM2 : UId AId AId AId -> Bool
op TERM' : AId AId AId -> Bool
op RESULT :  -> Bool

vars A A1 A2 : AId
var U : UId
vars N1 N2 M1 M2 : Nat

eq state-of-system(M1, M2) = add-user(u1, M1,
                      add-user(u2, M2, t)) .

eq w1w2(A1, A2, N1, N2, M1, M2) =
   withdraw(u1, A1, N1,
   withdraw(u2, A2, N2, state-of-system(M1, M2))) .

eq w2w1(A1, A2, N1, N2, M1, M2) =
   withdraw(u2, A2, N2,
   withdraw(u1, A1, N1, state-of-system(M1, M2))) .
```

Notice that the balances of the accounts are specified via the method `add-user`; this trick does not affect the generality of the proof.

The following sequence of equations gradually eliminates the parameters by instantiating them to constants describing the case analysis:

```
eq TERM(U, A, A1, A2, N1, N2, M1, M2) =
     w1w2(A1,A2,N1,N2,M1,M2) R[U, A]
     w2w1(A1,A2,N1,N2,M1,M2) .

eq TERM1(U, A, A1, A2, N2, M2) =
       TERM(U, A, A1, A2, n1, N2, m1, M2) and
       TERM(U, A, A1, A2, n1, N2, m1', M2) and
       TERM(U, A, A1, A2, n01, N2, m1, M2) .

eq TERM2 (U, A, A1, A2) =
       TERM1(U, A, A1, A2, n2, m2) and
       TERM1(U, A, A1, A2, n2, m2') and
       TERM1(U, A, A1, A2, n02, m2) .

eq TERM'(A, A1, A2) = TERM2(u,  A, A1, A2) and
                      TERM2(u1, A, A1, A2) and
                      TERM2(u2, A, A1, A2) .
eq RESULT = TERM'(a,  a1, a2) and
```

```
TERM'(a,  a1, a1) and
TERM'(a,  a,  a) and
TERM'(a1, a1, a2) .
```

The CafeOBJ system performs nearly 600,000 reductions for `RESULT` and gives `true`.

8 Conclusion and future work

We have presented a new method of object composition in the algebraic specification language CafeOBJ . Our method can reuse not only specification code but also the proofs of behavioural equivalence. If we have objects which are already proved as valid then we can reuse them to build incrementally the specification of the target system. We think that the reusability of proofs is the key technique to reduce the cost of verification especially in the specification languages having a support mechanism for proofs (theorem provers, proof checkers, etc). Also, our method supports class libraries for algebraic specification languages.

We have also made a comparison between our object composition approach and the inheritance approach (also in CafeOBJ). Proofs for behavioural equivalences can reuse in both composition and inheritance approach, but reusability is higher when in the composition approach.

In this paper, we have used OMT like notation to represent relations of objects. We plan to study further the relationship between the OMT-style specification and CafeOBJ-style specifications and study other object-oriented concepts within the CafeOBJ framework. Also, we plan to build a class library for CafeOBJ. We think that tools capable of handling our method, some graphical notations (such as OMT), and class libraries in an uniform environment are an important future research topic too.

References

[1] P. Borba and J.A. Goguen. Refinement of concurrent object-oriented programs. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.

[2] Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for CafeOBJ. Technical Report IS-RR-96-0024S, Japan Advanced Institute of Science and Technology (JAIST), 1996.

[3] Răzvan Diaconescu and Kokichi Futatsugi. CafeOBJ *Report*. AMAST. World Scientific, 1998. To appear.

[4] Roger Duke, Paul King, and Graeme Smith Gordon Rose. The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Center, Department of Computer Science, The University of Queensland, April 1991.

[5] Kokichi Futatsugi. An overview of cafe specification environment — an algebraic approach for creating, verifying and maintaining formal specifications over the net —. In *First IEEE International Conference on Formal Engineering Methods*. IEEE, 1997.

[6] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[7] Kokichi Futatsugi and Toshimi Sawada. Design considerations for Cafe specification environment. In *The 10th Anniversary of OBJ2*, October 1995.

[8] Joseph Goguen and Rod Burstall. Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1), 1992.

[9] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(4), 1994.

[10] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Harmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*. Springer, 1994.

[11] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, UCSD Technical Report, April 1997. http://www-cse.ucsd.edu/users/goguen/pubs/index.html.

[12] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2), 1992.

[13] Joseph Goguen, Timothy Winkler, José Mesegure, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, SRI International, Computer Science Laboratory, 1993.

[14] R. Hennicker. Context induction: a proof principle for behavioural abstractions. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems. International Symposium DISCO 1990*, number 429 in LNCS. Springer-Verlag, 1990.

[15] Shusaku Iida, Kokichi Futatsugi, and Takuo Watanabe. Algebraic specification of distributed systems based on concurrent object-oriented modeling. In Elie Najm and Jean-Bernard Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*. Chapman & Hall, 1996.

[16] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 93:pp.73–155, 1992.

[17] Ataru. T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. CafeOBJ user's manual, 1997. http://ldl.jaist.ac.jp:8080/cafeobj.

[18] James Rumbaugh and et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[19] Toshimi Sawada and Kokichi Futatsugi. Basic features of CHAOS specification kernel language. In *The 10th Anniversary of OBJ2*, October 1995.

A Proof of Theorem 6

Proof 1 *Consider two states s and s' such that for all composing objects O_n, $\pi_n(s) \equiv_n \pi_n(s')$. Consider a visible context c for O. We have to prove that $c(s) = c(s')$. We have two cases:*

1. *$c = m;\pi_n;c_n$ for some object n, a sequence m of O_n-methods, such that π_n is behavioural and c_n is a visible context for n.*

2. *$c = m;a$ for some sequence m of O-methods and an O-attribute a.*

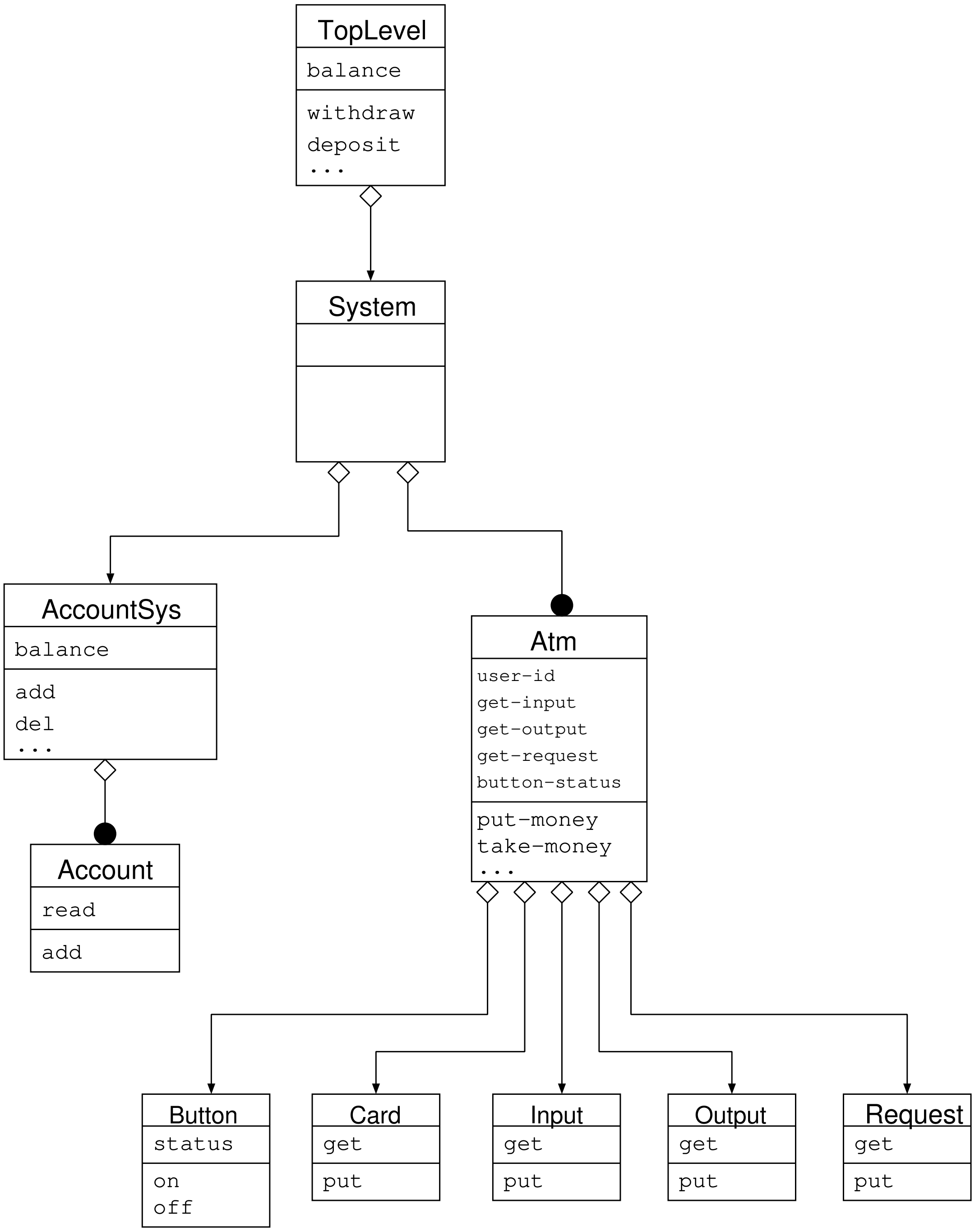
Let's concentrate first on the case 1. If m is empty (i.e., empty sequence) then we can apply directly the hypothesis. Therefore we may assume m is a single "atomic" method (the other case can be proved by iterating the "atomic" case). Because, all conditions cond of the equations having $m;\pi_n$ as the left hand side are equalities of terms which are either of the form $\pi_k;c_k$ (where O_k is a composing object and c_k is visible O_k-context) or terms in the signature of the data, we have $cond(s) = cond(s')$ for all these conditions. Therefore we pick the condition cond for which both $cond(s)$ and $cond(s')$ are true, and let $\pi_n;m_n$ be the right hand side of the corresponding equation. Then

$$\begin{aligned} c(s) = c(s') \quad & \textit{iff} \quad (m;\pi_n;c_n)(s) = (m;\pi_n;c_n)(s') \\ & \textit{iff} \quad (\pi_n;m_n;c_n)(s) = (\pi_n;m_n;c_n)(s') \end{aligned}$$

which holds because $m_n;c_n$ is a visible O_n-context and $\pi_n(s) \equiv_n \pi_n(s')$.

Finally, case 2 can be reduced to case 1 by a similar argument with the above, with the difference that π_n is not necessarily behavioural. □

B Whole structure of the ATM system



C A specification of the ATM system

```
-- -----------------------------------------------------------------
-- Values of SWITCH
-- -----------------------------------------------------------------
mod! ON-OFF {
  [ Value ]

  ops on off : -> Value
}

-- -----------------------------------------------------------------
-- SWITCH
-- -----------------------------------------------------------------
mod* SWITCH {
  protecting(ON-OFF)

  *[ Switch ]*

  op init-sw : -> Switch          -- initial state
  -- switch on
  bop on_ : Switch -> Switch      -- method
  -- switch off
  bop off_ : Switch -> Switch     -- method
  -- observe the state of the switch
  bop status_ : Switch -> Value    -- attribute

  var S : Switch

  eq status(init-sw) = off .
  eq status(on(S)) = on .
  eq status(off(S)) = off .
}

-- -----------------------------------------------------------------
-- User identification
-- -----------------------------------------------------------------
mod! USER-ID {
  protecting(NAT)
  [ Nat < UId ]

  op unidentified-user : -> UId
}

-- -----------------------------------------------------------------
-- Counter
-- -----------------------------------------------------------------
mod* COUNTER {
  protecting(USER-ID + INT)

  *[ Counter ]*

  -- initialize counter with user ID
  op init-counter : UId -> Counter    -- initial state
  -- add a value to the counter
  bop add : Int Counter -> Counter    -- method
  -- read the value of the counter
  bop read_ : Counter -> Int          -- attribute

  var I : Int
  var C : Counter
  var U : UId
```

```
  eq read(init-counter(U)) = 0 .
  eq read(add(I, C)) = I + read(C) .
}

-- ----------------------------------------------------------------
-- Counter with error
-- ----------------------------------------------------------------
mod* COUNTER* {
  protecting(COUNTER)

  -- error value
  op counter-not-exist : -> Counter  -- error
}

-- ----------------------------------------------------------------
-- Account system
-- ----------------------------------------------------------------
mod* ACCOUNT-SYSTEM {
  protecting(COUNTER* *{ hsort Counter -> Account,
                         op init-counter -> init-account,
                         op counter-not-exist -> no-account })

  *[ AccountSys ]*

  op init-account-sys : -> AccountSys             -- initial state
  -- add a user account with user ID
  bop add : UId Nat AccountSys -> AccountSys       -- method
  -- delete a user account
  bop del : UId AccountSys -> AccountSys           -- method
  -- deposit operation
  bop deposit : UId Nat AccountSys -> AccountSys   -- method
  -- withdraw operation
  bop withdraw : UId Nat AccountSys -> AccountSys  -- method
  -- calculate the balance of an user account
  bop balance : UId AccountSys -> Nat              -- attribute
  -- get the state of a counter from the state of an account
  bop account : UId AccountSys -> Account          -- projection

  vars U U' : UId
  var A : AccountSys
  var N : Nat

  eq account(U, init-account-sys) = no-account .
  ceq account(U, add(U', N, A)) = add(N, init-account(U))
      if U == U' .
  ceq account(U, add(U', N, A)) = account(U, A)
      if U =/= U' .
  ceq account(U, del(U', A)) = no-account
      if U == U' .
  ceq account(U, del(U', A)) = account(U, A)
      if U =/= U' .
  ceq account(U, deposit(U', N, A)) = add(N, account(U, A))
      if U == U' .
  ceq account(U, deposit(U', N, A)) = account(U, A)
      if U =/= U' .
  ceq account(U, withdraw(U', N, A)) = add(-(N), account(U, A))
      if U == U' .
  ceq account(U, withdraw(U', N, A)) = account(U, A)
      if U =/= U' .

  eq balance(U, A) = read(account(U, A)) .
}
```

```
-- -------------------------------------------------------------
-- Trivial module with an element (undefined)
-- -------------------------------------------------------------
mod* TRIV+ {
  [ Elt ]

  op undefined : -> Elt
}

-- -------------------------------------------------------------
-- Cell
-- -------------------------------------------------------------
mod* CELL(X :: TRIV+) {
  *[ Cell ]*

  op init-cell : -> Cell      -- initial state
  -- put the element to the cell
  bop put : Elt Cell -> Cell  -- method
  -- get the element from the cell
  bop get : Cell -> Elt       -- attribute

  var E : Elt
  var C : Cell

  eq get(init-cell) = undefined .
  eq get(put(E, C)) = E .
}

-- -------------------------------------------------------------
-- ATM identifier
-- -------------------------------------------------------------
mod! ATM-ID {
  protecting(NAT *{ sort Nat -> AId })
}

-- -------------------------------------------------------------
--  Button
-- -------------------------------------------------------------
mod* BUTTON {
  protecting(SWITCH *{ hsort Switch -> Button,
                       sort Value -> Operation,
                       op init-sw -> init-button,
                       op on -> deposit,
                       op off -> withdraw })
}

-- -------------------------------------------------------------
--  Cell for card information
-- -------------------------------------------------------------
mod* CARD {
  protecting(CELL(X <= view to USER-ID
                  { sort Elt -> UId,
                    op undefined -> unidentified-user })
                  *{ hsort Cell -> Card,
                     op init-cell -> init-card })
}

-- -------------------------------------------------------------
--  Cell for input
-- -------------------------------------------------------------
mod* INPUT {
  protecting(CELL(X <= view to NAT
                  { sort Elt -> Nat,
```

```
                  op undefined -> 0 })
                *{ hsort Cell -> Input,
                   op init-cell -> init-input })
}

-- --------------------------------------------------------------
--  Cell for output
-- --------------------------------------------------------------
mod* OUTPUT {
  protecting(CELL(X <= view to NAT
                { sort Elt -> Nat,
                  op undefined -> 0 })
                *{ hsort Cell -> Output,
                   op init-cell -> init-output })
}

-- --------------------------------------------------------------
--  Cell for request
-- --------------------------------------------------------------
mod* REQUEST {
  protecting(CELL(X <= view to NAT
                { sort Elt -> Nat,
                  op undefined -> 0 })
                *{ hsort Cell -> Request,
                   op init-cell -> init-request })
}

-- --------------------------------------------------------------
--  ATM client
-- --------------------------------------------------------------
mod* ATM-CLIENT {
-- importing data and the composing objects
  protecting(ATM-ID + BUTTON + CARD + INPUT + OUTPUT + REQUEST)

  *[ Atm ]*

  op init-atm : AId -> Atm              -- initial state
  op no-atm : -> Atm                    -- error
  op invalid-operation : -> Atm         -- error
  -- push the deposit button
  bop deposit : Atm -> Atm              -- method
  -- push the withdraw button
  bop withdraw : Atm -> Atm             -- method
  -- input the request for withdraw
  bop request : Nat Atm -> Atm          -- method
  -- put money
  bop put-money : Nat Atm -> Atm        -- method
  -- take money
  bop take-money : Atm -> Atm           -- method
  -- set money for output (system operation)
  bop set-money : Nat Atm -> Atm        -- method
  -- put the bank card
  bop put-card : UId Atm -> Atm         -- method
  -- clear all the informations kept in the atm
  bop clear : Atm -> Atm                -- method
  -- get the user ID
  bop user-id : Atm -> UId              -- attribute
  -- get the money that user input
  bop get-input : Atm -> Nat            -- attribute
  -- get the outputed money
  bop get-output : Atm -> Nat           -- attribute
  -- get the request
  bop get-request : Atm -> Nat          -- attribute
```

```
-- get the state of the button
bop button-status : Atm -> Operation  -- attribute

bop button : Atm -> Button            -- projection
bop card : Atm -> Card                -- projection
bop request : Atm -> Request          -- projection
bop input : Atm -> Input              -- projection
bop output : Atm -> Output            -- projection

var ATM : Atm
var N : Nat
var U : UId
var A : AId

eq button(init-atm(A)) = init-button .
eq button(invalid-operation) = init-button .
eq button(deposit(ATM)) = on(button(ATM)) .
eq button(withdraw(ATM)) = off(button(ATM)) .
eq button(request(N, ATM)) = button(ATM) .
eq button(put-money(N, ATM)) = button(ATM) .
eq button(take-money(ATM)) = button(ATM) .
eq button(set-money(N, ATM)) = button(ATM) .
eq button(put-card(U, ATM)) = button(ATM) .
eq button(clear(ATM)) = init-button .

eq card(init-atm(A)) = init-card .
eq card(invalid-operation) = init-card .
eq card(deposit(ATM)) = card(ATM) .
eq card(withdraw(ATM)) = card(ATM) .
eq card(request(N, ATM)) = card(ATM) .
eq card(put-money(N, ATM)) = card(ATM) .
eq card(take-money(ATM)) = card(ATM) .
eq card(set-money(N, ATM)) = card(ATM) .
eq card(put-card(U, ATM)) = put(U, card(ATM)) .
eq card(clear(ATM)) = init-card .

eq request(init-atm(A)) = init-request .
eq request(invalid-operation) = init-request .
eq request(deposit(ATM)) = request(ATM) .
eq request(withdraw(ATM)) = request(ATM) .
eq request(request(N, ATM)) = put(N, request(ATM)) .
eq request(put-money(N, ATM)) = request(ATM) .
eq request(take-money(ATM)) = request(ATM) .
eq request(set-money(N, ATM)) = request(ATM) .
eq request(put-card(U, ATM)) = request(ATM) .
eq request(clear(ATM)) = init-request .

eq input(init-atm(A)) = init-input .
eq input(invalid-operation) = init-input .
eq input(deposit(ATM)) = input(ATM) .
eq input(withdraw(ATM)) = input(ATM) .
eq input(request(N, ATM)) = input(ATM) .
eq input(put-money(N, ATM)) = put(N, input(ATM)) .
eq input(take-money(ATM)) = input(ATM) .
eq input(set-money(N, ATM)) = input(ATM) .
eq input(put-card(U, ATM)) = input(ATM) .
eq input(clear(ATM)) = init-input .

eq output(init-atm(A)) = init-output .
eq output(invalid-operation) = init-output .
eq output(deposit(ATM)) = output(ATM) .
eq output(withdraw(ATM)) = output(ATM) .
eq output(request(N, ATM)) = output(ATM) .
```

```
  eq output(put-money(N, ATM)) = output(ATM) .
  eq output(take-money(ATM)) = init-output .
  eq output(set-money(N, ATM)) = put(N, output(ATM)) .
  eq output(put-card(U, ATM)) = output(ATM) .
  eq output(clear(ATM)) = output(ATM) .

  eq user-id(ATM) = get(card(ATM)) .
  eq get-input(ATM) = get(input(ATM)) .
  eq get-output(ATM) = get(output(ATM)) .
  eq get-request(ATM) = get(request(ATM)) .
  eq button-status(ATM) = status(button(ATM)) .
}

-- -----------------------------------------------------------------
-- ATM system
-- -----------------------------------------------------------------
mod* ATM-SYSTEM {
  protecting(ACCOUNT-SYSTEM + ATM-CLIENT)

  *[ System ]*

  op init-sys : -> System                     -- initial state
  -- add an atm to the system
  bop add-atm : AId System -> System          -- method
  -- delete an atm from the system
  bop del-atm : AId System -> System          -- method
  -- add an user account
  bop add-user : UId Nat System -> System     -- method
  -- delete an user account
  bop del-user : UId System -> System         -- method
  -- put the bank card
  bop put-card : AId UId System -> System     -- method
  -- request for withdraw
  bop request : AId Nat System -> System      -- method
  -- put money
  bop put-money : AId Nat System -> System    -- method
  -- take money
  bop take-money : AId System -> System       -- method
  -- deposit operation
  bop deposit : AId System -> System          -- method
  -- withdraw operation
  bop withdraw : AId System -> System         -- method
  -- push the ok button on atm to complete the operation
  bop ok : AId System -> System               -- method
  -- cancel the operation of ATM
  bop cancel : AId System -> System           -- method
  -- get the balance of specified user
  bop balance : UId System -> Nat             -- attribute
  -- projection operator for AccountSys
  bop account-sys : System -> AccountSys      -- projection
  -- projection operator for Atm
  bop atm : AId System -> Atm                 -- projection

  var S : System
  vars A A' : AId
  var U : UId
  var N : Nat

  eq balance(U, S) = balance(U, account-sys(S)) .

  eq account-sys(init-sys) = init-account-sys .
  eq account-sys(add-atm(A, S)) = account-sys(S) .
  eq account-sys(del-atm(A, S)) = account-sys(S) .
```

```
  eq account-sys(add-user(U, N, S)) = add(U, N, account-sys(S)) .
  eq account-sys(del-user(U, S)) = del(U, account-sys(S)) .
  eq account-sys(put-card(A, U, S)) = account-sys(S) .
  eq account-sys(request(A, N, S)) = account-sys(S) .
  eq account-sys(put-money(A, N, S)) = account-sys(S) .
  eq account-sys(take-money(A, S)) = account-sys(S) .
  eq account-sys(deposit(A, S)) = account-sys(S) .
  eq account-sys(withdraw(A, S)) = account-sys(S) .
  ceq account-sys(ok(A, S)) =
       deposit(user-id(atm(A, S)), get-input(atm(A, S)), account-sys(S))
       if button-status(atm(A, S)) == deposit and
          user-id(atm(A, S)) =/= unidentified-user and
          get-input(atm(A, S)) =/= 0 .
  ceq account-sys(ok(A, S)) =
       withdraw(user-id(atm(A, S)), get-request(atm(A, S)), account-
sys(S))
       if button-status(atm(A, S)) == withdraw and
          user-id(atm(A, S)) =/= unidentified-user and
          get-request(atm(A, S)) =/= 0 and
          get-request(atm(A, S)) <=
                 balance(user-id(atm(A, S)), account-sys(S)) .
  ceq account-sys(ok(A, S)) = account-sys(S)
       if user-id(atm(A, S)) == unidentified-user or
          (button-status(atm(A, S)) == deposit and
              get-input(atm(A, S)) == 0) or
          (button-status(atm(A, S)) == withdraw and
              (get-request(atm(A, S)) == 0 or
              get-request(atm(A, S)) >
                       balance(user-id(atm(A, S)), account-sys(S)))) .
  eq account-sys(cancel(A, S)) = account-sys(S) .

  eq atm(A, init-sys) = no-atm .
  ceq atm(A, add-atm(A', S)) = init-atm(A)
       if A == A' .
  ceq atm(A, add-atm(A', S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, del-atm(A', S)) = no-atm
       if A == A .
  ceq atm(A, del-atm(A', S)) = atm(A, S)
       if A =/= A .
  eq atm(A, add-user(U, N, S)) = atm(A, S) .
  eq atm(A, del-user(U, S)) = atm(A, S) .
  ceq atm(A, put-card(A', U, S)) = put-card(U, atm(A, S))
       if A == A' .
  ceq atm(A, put-card(A', U, S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, request(A', N, S)) = request(N, atm(A, S))
       if A == A' .
  ceq atm(A, request(A', N, S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, put-money(A', N, S)) = put-money(N, atm(A, S))
       if A == A' .
  ceq atm(A, put-money(A', N, S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, take-money(A', S)) = take-money(atm(A, S))
       if A == A' .
  ceq atm(A, take-money(A', S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, deposit(A', S)) = deposit(atm(A, S))
       if A == A' .
  ceq atm(A, deposit(A', S)) = atm(A, S)
       if A =/= A' .
  ceq atm(A, withdraw(A', S)) = withdraw(atm(A, S))
```

```
      if A == A' .
  ceq atm(A, withdraw(A', S)) = atm(A, S)
      if A =/= A' .
  ceq atm(A, ok(A', S)) = clear(atm(A, S))
      if A == A' and
         user-id(atm(A, S)) =/= unidentified-user and
         button-status(atm(A, S)) == deposit .
  ceq atm(A, ok(A', S)) = set-money(get-request(atm(A, S)), clear(atm(A, S)))
      if A == A' and
         user-id(atm(A, S)) =/= unidentified-user and
         button-status(atm(A, S)) ==  withdraw and
         get-request(atm(A, S)) <=
                   balance(user-id(atm(A, S)), account-sys(S)) .
  ceq atm(A, ok(A', S)) = invalid-operation
      if A == A' and
         (user-id(atm(A, S)) == unidentified-user or
         (button-status(atm(A, S)) ==  withdraw and
             (get-request(atm(A, S)) >
                  balance(user-id(atm(A, S)), account-sys(S))))) .
  ceq atm(A, ok(A', S)) = atm(A, S)
      if A =/= A' .
  ceq atm(A, cancel(A', S)) = init-atm(A)
      if A == A' .
  ceq atm(A, cancel(A', S)) = atm(A, S)
      if A =/= A' .
}

-- ---------------------------------------------------------------
-- The toplevel of ATM system
-- ---------------------------------------------------------------
mod* ATM-SYSTEM-TOPLEVEL {
  protecting(ATM-SYSTEM)

  *[ TopLevel ]*

  op init-tl : -> TopLevel                          -- initial state
  -- add a new atm
  bop add-atm : AId TopLevel -> TopLevel            -- method
  -- delete an atm
  bop del-atm : AId TopLevel -> TopLevel            -- method
  -- create an user account with initial balance
  bop add-user : UId Nat TopLevel -> TopLevel       -- method
  -- delete an user account
  bop del-user : UId TopLevel -> TopLevel           -- method
  -- user "UId" goes to an ATM "AId" and deposit "Nat"
  bop deposit : UId AId Nat TopLevel -> TopLevel    -- method
  -- user "UId" goes to an ATM "AId" and withdraw "Nat"
  bop withdraw : UId AId Nat TopLevel -> TopLevel   -- method
  -- get a balance for the user
  bop balance : UId TopLevel -> Nat                 -- attribute
  -- projection operator for System
  bop system : TopLevel -> System                   -- projection

  var U : UId
  var A : AId
  var N : Nat
  var TL : TopLevel

  eq balance(U, TL) = balance(U, account-sys(system(TL))) .

  eq system(init-tl) = init-sys .
  eq system(add-atm(A, TL)) = add-atm(A, system(TL)) .
  eq system(del-atm(A, TL)) = del-atm(A, system(TL)) .
```

```
  eq system(add-user(U, N, TL)) = add-user(U, N, system(TL)) .
  eq system(del-user(U, TL)) = del-user(U, system(TL)) .
  eq system(deposit(U, A, N, TL)) =
     ok(A, put-money(A, N, deposit(A, put-card(A, U, system(TL))))) .
  eq system(withdraw(U, A, N, TL)) =
     take-money(A, ok(A, request(A, N, withdraw(A,
                put-card(A, U, system(TL)))))) .
}


-- ----------------------------------------------------------------
-- test for ATM-SYSTEM-TOPLEVEL
-- ----------------------------------------------------------------
open ATM-SYSTEM-TOPLEVEL

ops n1 n2 n3 : -> Nat .
ops u1 u2 : -> UId .
ops ai1 ai2 : -> AId .

red balance(u1, deposit(u1, ai1, 20,
            add-user(u1, 100, add-atm(ai1, init-tl)))) .
red balance(u1, withdraw(u1, ai1, 20, withdraw(u2, ai1, 30,
            add-user(u1, 100, add-user(u2, 100, add-atm(ai1, init-tl)))))) .
close

-- ----------------------------------------------------------------
-- module for behavioural equivalences
-- ----------------------------------------------------------------
mod COINDUCTION-REL {
  protecting(ATM-SYSTEM-TOPLEVEL)

-- behavioural equivalence for AccountSys
  op _R[_]_ : AccountSys UId AccountSys -> Bool {coherent}

  vars AS1 AS2 : AccountSys
  var U : UId

  eq AS1 R[U] AS2 = account(U, AS1) =*= account(U, AS2) .

-- behavioural equivalence for Atm
  op _R_ : Atm Atm -> Bool {coherent}

  vars A1 A2 : Atm

  eq A1 R A2 = button(A1) =*= button(A2) and
               card(A1) =*= card(A2) and
               request(A1) =*= request(A2) and
               input(A1) =*= input(A2) and
               output(A1) =*= output(A2) .

-- behavioural equivalence for System
  op _R[_,_]_ : System UId AId System -> Bool {coherent}

  vars S1 S2 : System
  var A : AId

  eq S1 R[U, A] S2 = account-sys(S1) R[U] account-sys(S2) and
                     atm(A, S1) R atm(A, S2) .

-- behavioural equivalence for TopLevel
  op _R[_,_]_ : TopLevel UId AId TopLevel -> Bool {coherent}

  vars T1 T2 : TopLevel
```

```
  eq T1 R[U, A] T2 = system(T1) R[U, A] system(T2) .
}

mod PROOF {
  protecting(COINDUCTION-REL)

  ops a a1 a2 : -> AId
  op t : -> TopLevel
  ops u u1 u2 : -> UId
  ops n1 n2 n01 n02 m1 m1' m2 m2' : -> Nat

  eq n1 =/= 0 = true .
  eq n2 =/= 0 = true .
  eq n01 == 0 = true .
  eq n02 == 0 = true .
  eq n1 <= m1 = true .
  eq n01 <= m1 = true .
  eq n1 > m1' = true .
  eq n2 <= m2 = true .
  eq n02 <= m2 = true .
  eq n2 > m2' = true .

  op state-of-system : Nat Nat -> TopLevel
  ops w1w2 w2w1 : AId AId Nat Nat Nat Nat -> TopLevel
  op TERM : UId AId AId AId Nat Nat Nat Nat -> Bool
  op TERM1 : UId AId AId AId Nat Nat -> Bool
  op TERM2 : UId AId AId AId -> Bool
  op TERM' : AId AId AId -> Bool
  op RESULT :  -> Bool

  vars A A1 A2 : AId
  var U : UId
  vars N1 N2 M1 M2 : Nat

  eq state-of-system(M1, M2) = add-user(u1, M1,
                         add-user(u2, M2,
                         add-atm(a, t))) .

  eq w1w2(A1, A2, N1, N2, M1, M2) =
             withdraw(u1, A1, N1,
             withdraw(u2, A2, N2, state-of-system(M1, M2))) .

  eq w2w1(A1, A2, N1, N2, M1, M2) =
             withdraw(u2, A2, N2,
             withdraw(u1, A1, N1, state-of-system(M1, M2))) .

  eq TERM(U, A, A1, A2, N1, N2, M1, M2) =
       w1w2(A1,A2,N1,N2,M1,M2) R[U, A] w2w1(A1,A2,N1,N2,M1,M2) .

  eq TERM1(U, A, A1, A2, N2, M2) =
         TERM(U, A, A1, A2, n1, N2, m1, M2) and
 TERM(U, A, A1, A2, n1, N2, m1', M2) and
 TERM(U, A, A1, A2, n01, N2, m1, M2) .

  eq TERM2 (U, A, A1, A2) =
         TERM1(U, A, A1, A2, n2, m2) and
 TERM1(U, A, A1, A2, n2, m2') and
 TERM1(U, A, A1, A2, n02, m2) .

  eq TERM'(A, A1, A2) = TERM2(u,  A, A1, A2) and
                        TERM2(u1, A, A1, A2) and
TERM2(u2, A, A1, A2) .
```

```
  eq RESULT = TERM'(a,  a1, a2) and
              TERM'(a,  a1, a1) and
              TERM'(a,  a,  a) and
              TERM'(a1, a1, a2) .
}
select PROOF .
red RESULT .
```